

# AI ASSISTED CODING LAB

## ASSIGNMENT-8.2

ENROLLMENT NO :2503A51L20

BATCH NO: 19

NAME: Guangsinlung Phaomei

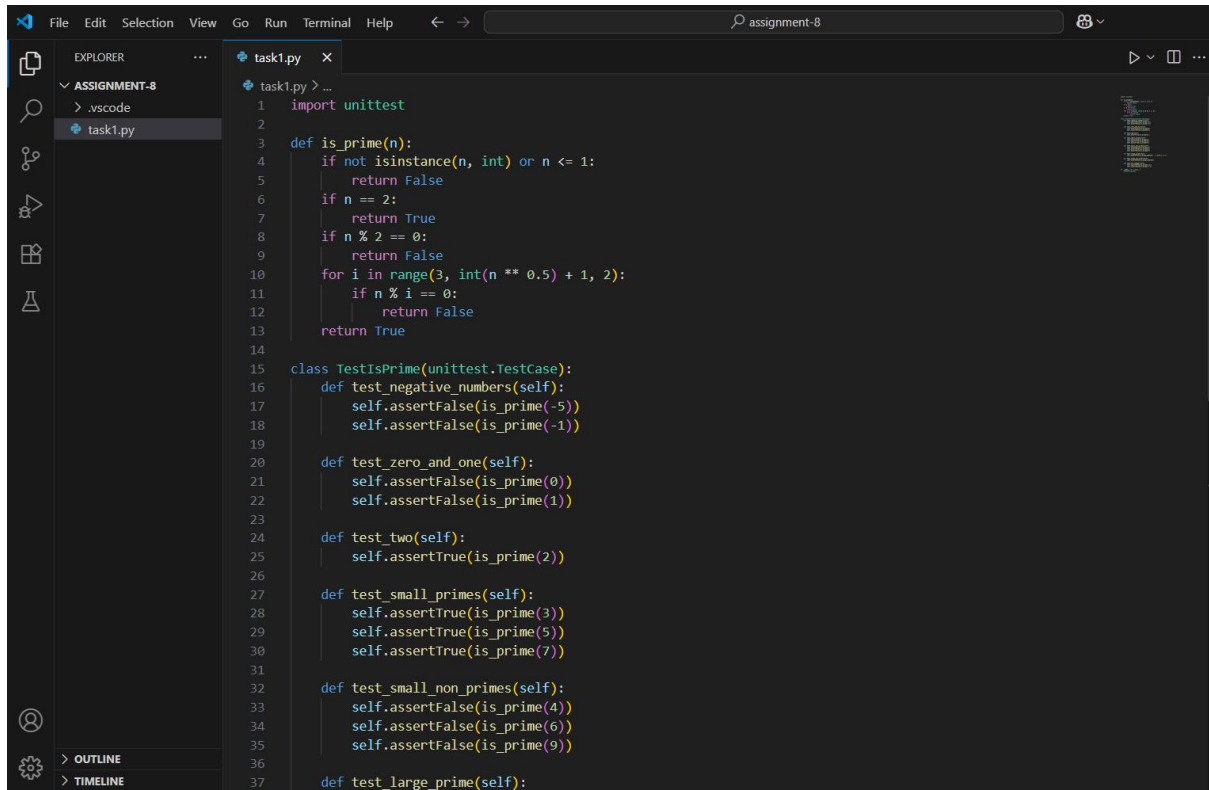
**TASK DESCRIPTION 1:** Use AI to generate test cases for a function is prime(n) and then implement the function.

### Requirements:

- Only integers  $> 1$  can be prime.
- Check edge cases: 0, 1, 2, negative numbers, and large primes

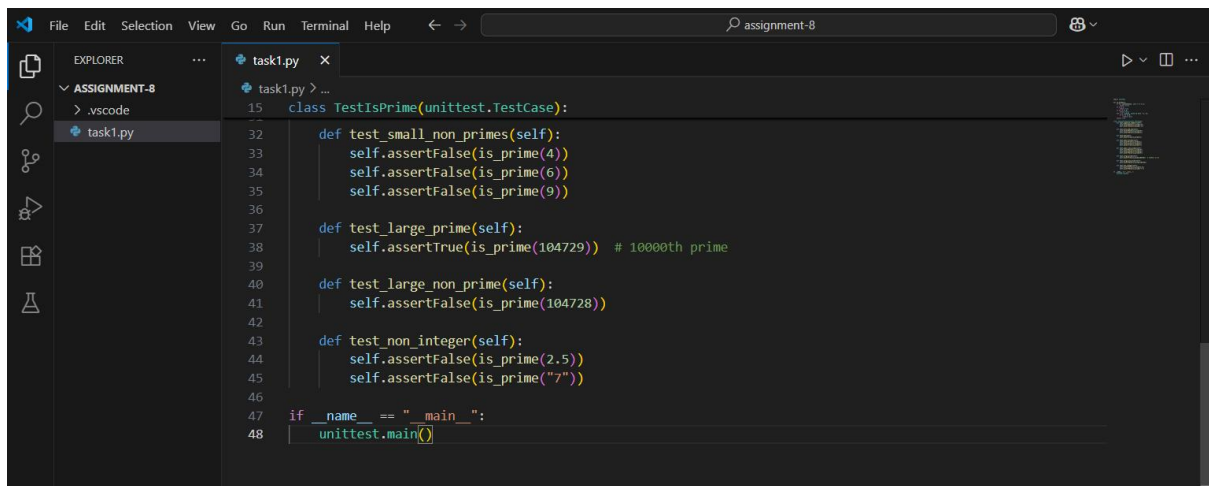
**PROMPT 1:** Begin by writing a Python function named is prime(n) that determines whether a given integer is a prime number. Use AI to generate diverse test cases for this function, including edge cases such as 0, 1, 2, negative numbers, and large prime numbers. Ensure that only integers greater than 1 are considered prime. After generating the test cases, implement the is prime(n) function and validate it using the generated test cases.

# CODE GENERATED:



This screenshot shows the first 37 lines of the Python file `task1.py` in a VS Code editor. The Explorer sidebar on the left shows the project structure with `task1.py` selected. The code defines a function `is_prime` and a test class `TestIsPrime`.

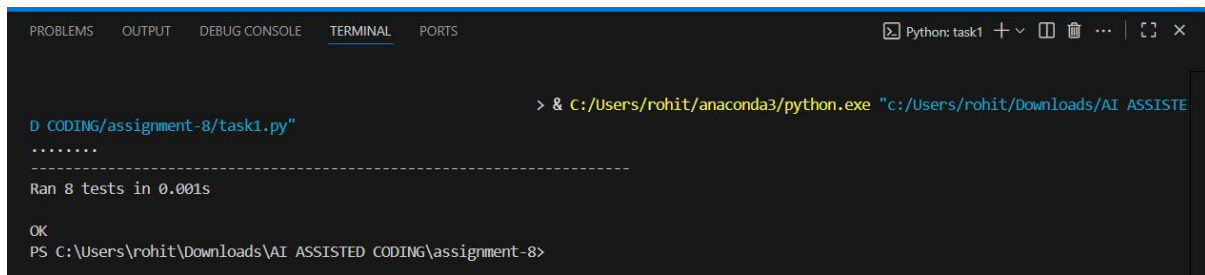
```
1 import unittest
2
3 def is_prime(n):
4     if not isinstance(n, int) or n <= 1:
5         return False
6     if n == 2:
7         return True
8     if n % 2 == 0:
9         return False
10    for i in range(3, int(n ** 0.5) + 1, 2):
11        if n % i == 0:
12            return False
13    return True
14
15 class TestIsPrime(unittest.TestCase):
16     def test_negative_numbers(self):
17         self.assertFalse(is_prime(-5))
18         self.assertFalse(is_prime(-1))
19
20     def test_zero_and_one(self):
21         self.assertFalse(is_prime(0))
22         self.assertFalse(is_prime(1))
23
24     def test_two(self):
25         self.assertTrue(is_prime(2))
26
27     def test_small_primes(self):
28         self.assertTrue(is_prime(3))
29         self.assertTrue(is_prime(5))
30         self.assertTrue(is_prime(7))
31
32     def test_small_non_primes(self):
33         self.assertFalse(is_prime(4))
34         self.assertFalse(is_prime(6))
35         self.assertFalse(is_prime(9))
36
37     def test_large_prime(self):
```



This screenshot shows the continuation of the Python file `task1.py` in a VS Code editor, starting from line 15. The code continues the `TestIsPrime` class with more test methods and a main execution block.

```
15 class TestIsPrime(unittest.TestCase):
16
17     def test_small_non_primes(self):
18         self.assertFalse(is_prime(4))
19         self.assertFalse(is_prime(6))
20         self.assertFalse(is_prime(9))
21
22     def test_large_prime(self):
23         self.assertTrue(is_prime(104729)) # 10000th prime
24
25     def test_large_non_prime(self):
26         self.assertFalse(is_prime(104728))
27
28     def test_non_integer(self):
29         self.assertFalse(is_prime(2.5))
30         self.assertFalse(is_prime("7"))
31
32 if __name__ == "__main__":
33     unittest.main()
```

## OUTPUT:

A screenshot of a terminal window with a dark background. The terminal shows the command to run a Python script: `> & C:/Users/rohit/anaconda3/python.exe "c:/Users/rohit/Downloads/AI ASSISTED CODING/assignment-8/task1.py"`. The output indicates that 8 tests were run successfully in 0.001 seconds. The terminal window has tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (selected), and PORTS. The title bar shows 'Python: task1' and standard window controls.

**OBSERVATION:** The task focuses on using AI to automatically generate meaningful test cases for the `is_prime(n)` function, ensuring that both typical and edge scenarios are covered. Test cases include inputs like 0, 1, negative numbers, small primes such as 2 and 3, as well as large prime numbers. This helps in validating the correctness and robustness of the implementation. The function `is_prime(n)` was then developed to correctly identify prime numbers by ensuring that only integers greater than 1 are considered and by applying logical checks for divisibility. Running the AI-generated test cases confirmed that the function works as expected for normal inputs and edge cases.

**TASK DESCRIPTION 2:** Ask AI to generate test cases for `celsius_to_fahrenheit(c)` and `fahrenheit_to_celsius(f)`.

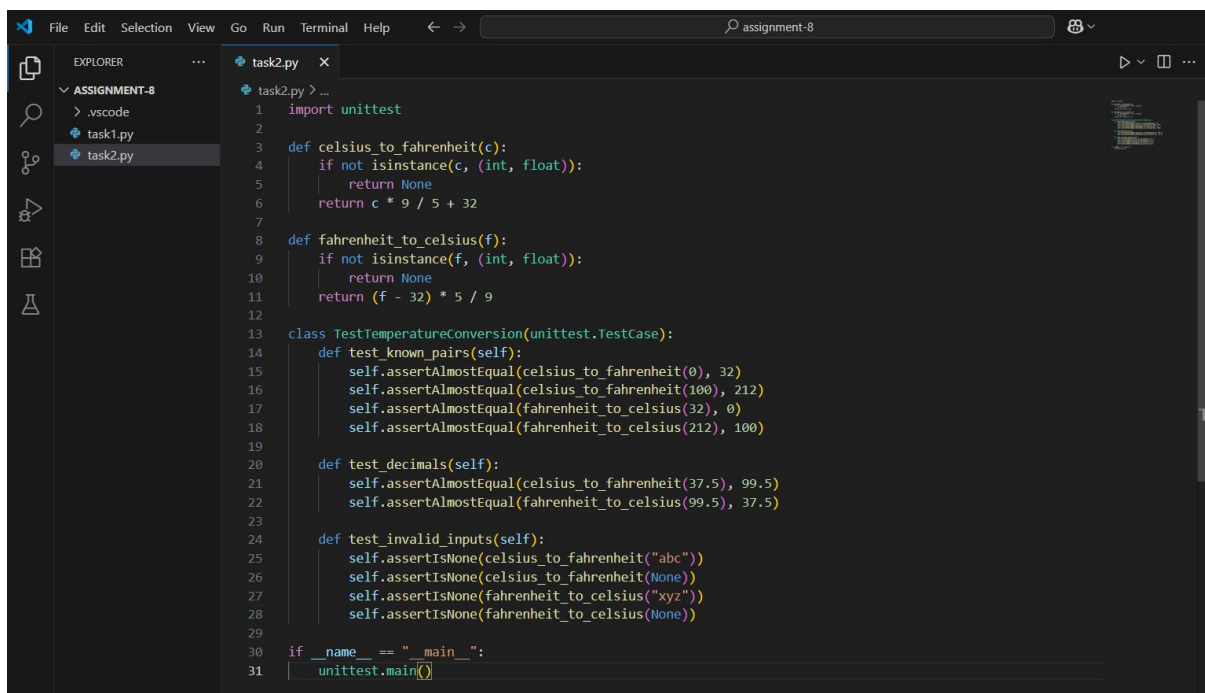
### Requirements:

- Validate known pairs:  $0^{\circ}\text{C} = 32^{\circ}\text{F}$ ,  $100^{\circ}\text{C} = 212^{\circ}\text{F}$ .
- Include decimals and invalid inputs like strings or None

**PROMPT 1 :** Write two Python functions: `celsius_to_fahrenheit(c)` and `fahrenheit_to_celsius(f)` for temperature conversion. Ask AI to generate

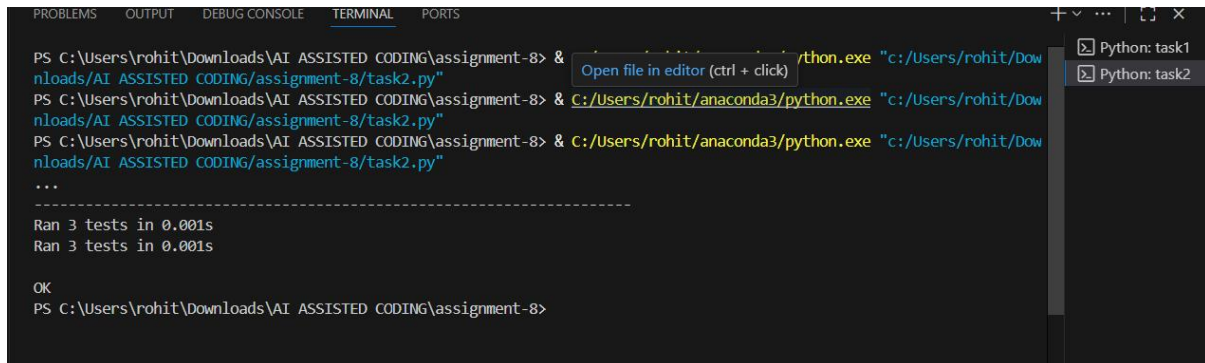
diverse test cases to validate these functions. The test cases should include known conversion pairs such as  $0^{\circ}\text{C} = 32^{\circ}\text{F}$  and  $100^{\circ}\text{C} = 212^{\circ}\text{F}$ , decimal values to test precision, and invalid inputs like strings or None to check error handling. After generating the test cases, implement both functions and verify their correctness against the test cases.

## CODE GENERATED :



```
1 import unittest
2
3 def celsius_to_fahrenheit(c):
4     if not isinstance(c, (int, float)):
5         return None
6     return c * 9 / 5 + 32
7
8 def fahrenheit_to_celsius(f):
9     if not isinstance(f, (int, float)):
10        return None
11    return (f - 32) * 5 / 9
12
13 class TestTemperatureConversion(unittest.TestCase):
14     def test_known_pairs(self):
15         self.assertEqual(celsius_to_fahrenheit(0), 32)
16         self.assertEqual(celsius_to_fahrenheit(100), 212)
17         self.assertEqual(fahrenheit_to_celsius(32), 0)
18         self.assertEqual(fahrenheit_to_celsius(212), 100)
19
20     def test_decimals(self):
21         self.assertEqual(celsius_to_fahrenheit(37.5), 99.5)
22         self.assertEqual(fahrenheit_to_celsius(99.5), 37.5)
23
24     def test_invalid_inputs(self):
25         self.assertIsNone(celsius_to_fahrenheit("abc"))
26         self.assertIsNone(celsius_to_fahrenheit(None))
27         self.assertIsNone(fahrenheit_to_celsius("xyz"))
28         self.assertIsNone(fahrenheit_to_celsius(None))
29
30 if __name__ == "__main__":
31     unittest.main()
```

## OUTPUT :



```
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8> & /thon.exe "c:/Users/rohit/Dow
nloads/AI ASSISTED CODING/assignment-8/task2.py"
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8> & C:/Users/rohit/anaconda3/python.exe "c:/Users/rohit/Dow
nloads/AI ASSISTED CODING/assignment-8/task2.py"
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8> & C:/Users/rohit/anaconda3/python.exe "c:/Users/rohit/Dow
nloads/AI ASSISTED CODING/assignment-8/task2.py"
...
-----
Ran 3 tests in 0.001s
Ran 3 tests in 0.001s

OK
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8>
```

**OBSERVAION :** In this task, AI was used to generate a wide range of test cases to validate the correctness of the two temperature conversion functions `celsius_to_fahrenheit(c)` and `fahrenheit_to_celsius(f)`. The generated cases included standard known values such as  $0^{\circ}\text{C} = 32^{\circ}\text{F}$  and  $100^{\circ}\text{C} = 212^{\circ}\text{F}$ , which helped confirm the accuracy of the conversion formulas. Decimal values were also tested to ensure precision in fractional conversions. Additionally, invalid inputs such as strings and `None` were included to verify the robustness and error-handling capability of the functions. The implementation successfully passed the valid test cases while appropriately handling invalid inputs, demonstrating correctness, precision, and reliability.

**TASK DESCRIPTION 3:** Use AI to write test cases for a function `count_words(text)` that returns the number of words in a sentence.

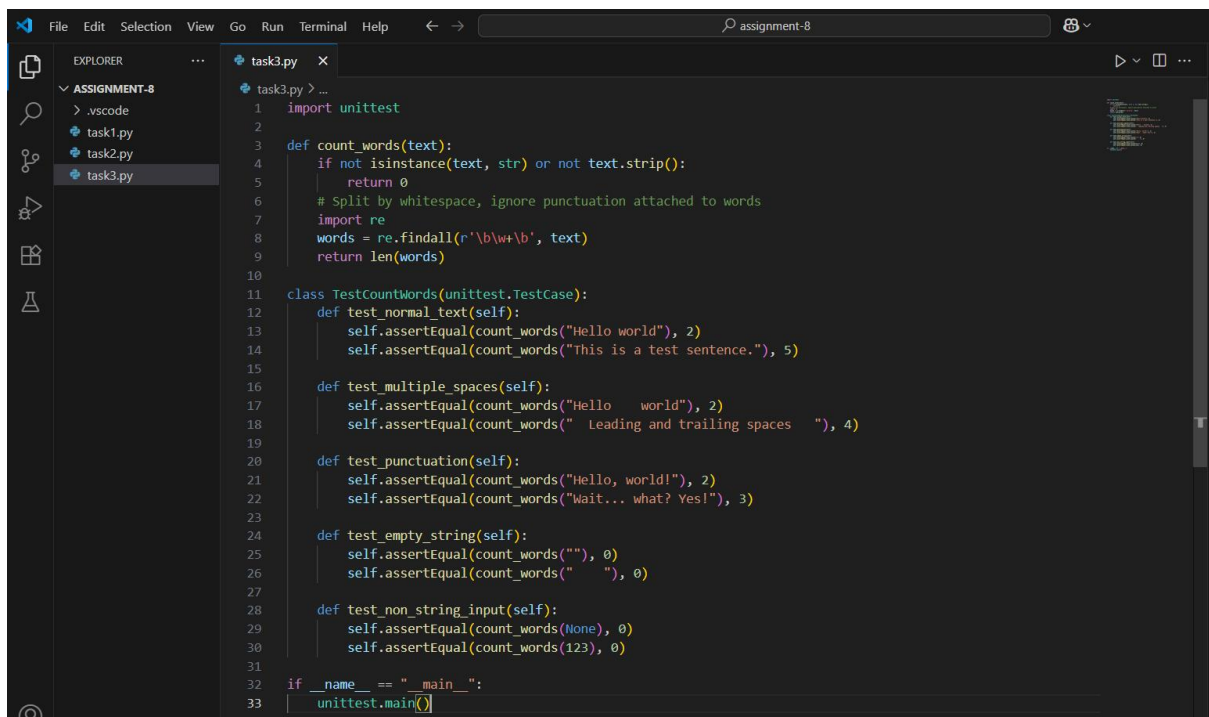
### Requirement:

Handle normal text, multiple spaces, punctuation, and empty strings

## PROMPT 1:

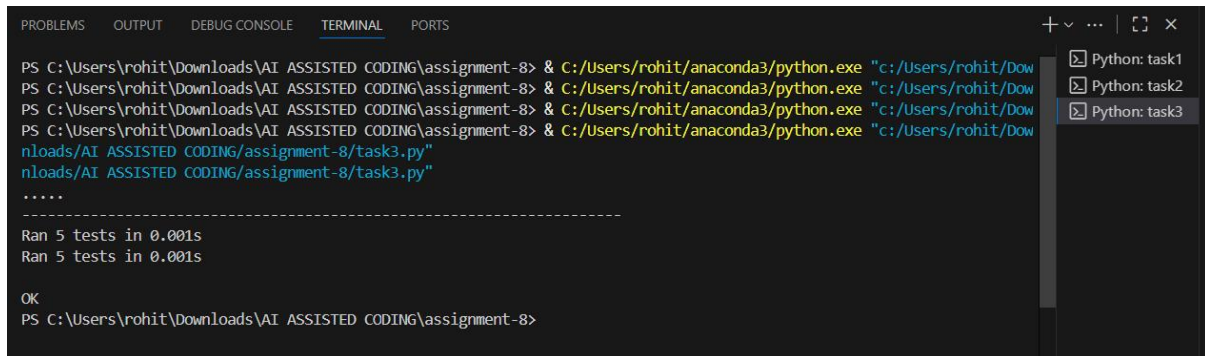
Write a Python function `count_words(text)` that returns the number of words in a given sentence. Use AI to generate diverse test cases for this function. The test cases should cover normal sentences, inputs with multiple spaces, punctuation marks, and empty strings. After generating the test cases, implement the `count_words(text)` function and validate it using the generated cases."

## CODE GENERATED :



```
1  import unittest
2
3  def count_words(text):
4      if not isinstance(text, str) or not text.strip():
5          return 0
6      # Split by whitespace, ignore punctuation attached to words
7      import re
8      words = re.findall(r'\b\w+\b', text)
9      return len(words)
10
11 class TestCountWords(unittest.TestCase):
12     def test_normal_text(self):
13         self.assertEqual(count_words("Hello world"), 2)
14         self.assertEqual(count_words("This is a test sentence."), 5)
15
16     def test_multiple_spaces(self):
17         self.assertEqual(count_words("Hello   world"), 2)
18         self.assertEqual(count_words("   Leading and trailing spaces   "), 4)
19
20     def test_punctuation(self):
21         self.assertEqual(count_words("Hello, world!"), 2)
22         self.assertEqual(count_words("Wait... what? Yes!"), 3)
23
24     def test_empty_string(self):
25         self.assertEqual(count_words(""), 0)
26         self.assertEqual(count_words("   "), 0)
27
28     def test_non_string_input(self):
29         self.assertEqual(count_words(None), 0)
30         self.assertEqual(count_words(123), 0)
31
32 if __name__ == "__main__":
33     unittest.main()
```

## OUTPUT :



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8> & C:/Users/rohit/anaconda3/python.exe "c:/Users/rohit/Dow
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8> & C:/Users/rohit/anaconda3/python.exe "c:/Users/rohit/Dow
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8> & C:/Users/rohit/anaconda3/python.exe "c:/Users/rohit/Dow
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8> & C:/Users/rohit/anaconda3/python.exe "c:/Users/rohit/Dow
nloads/AI ASSISTED CODING/assignment-8/task3.py"
nloads/AI ASSISTED CODING/assignment-8/task3.py"
.....
Ran 5 tests in 0.001s
Ran 5 tests in 0.001s
OK
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8>
```

## OBSERVATION :

In this task, AI was utilized to generate comprehensive test cases for the `count_words(text)` function, which calculates the number of words in a given sentence. The test cases included normal sentences, inputs with multiple spaces, text containing punctuation, and empty strings. These variations ensured that the function was tested for both common and edge scenarios. The implementation of the function successfully handled all cases, correctly counting words despite irregular spacing or punctuation, and returning zero for empty strings. This demonstrates that the function is reliable, accurate, and robust across different input formats

**TASK DESCRIPTION 4:** Generate test cases for a `BankAccount` class with:

## Methods:

deposit(amount)  
withdraw(amount)  
check\_balance()

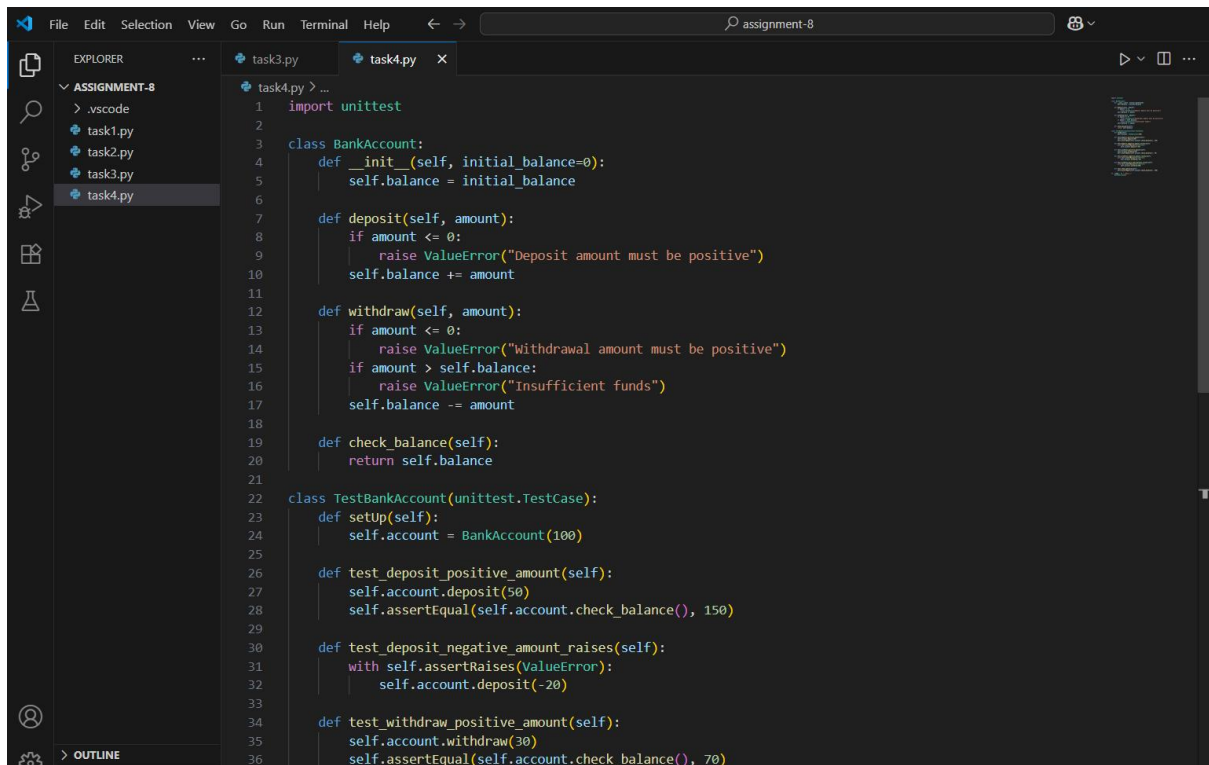
## Requirements:

- Negative deposits/withdrawals should raise an error.
- Cannot withdraw more than balance.

**PROMPT 1:** Write a Python class BankAccount with methods deposit(amount), withdraw(amount), and check\_balance(). Use AI to generate diverse test cases for this class. The test cases should include valid deposits and withdrawals, as well as invalid scenarios such as negative deposits, negative withdrawals, and attempts to withdraw more than the available balance. After generating the test cases, implement the BankAccount class and validate its behavior against the test cases."

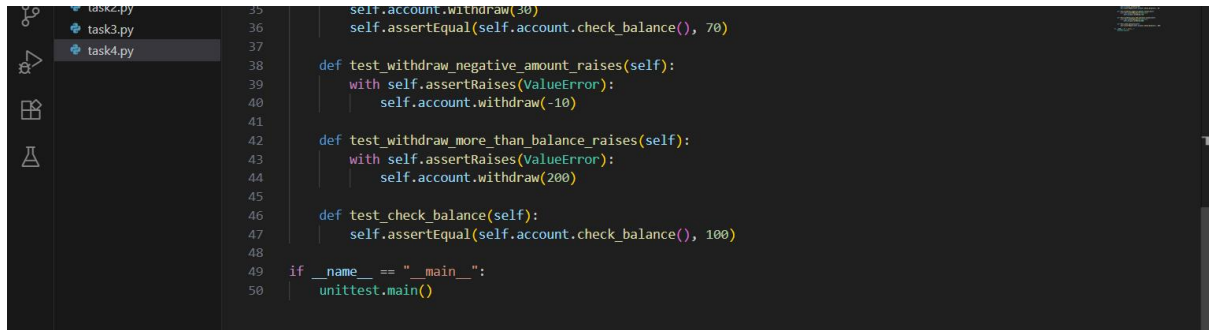


# CODE GENERATED:



The screenshot shows the Visual Studio Code editor with the file explorer on the left displaying a project named 'ASSIGNMENT-8' containing files task1.py, task2.py, task3.py, and task4.py. The main editor window shows the code for task4.py, which includes a BankAccount class and a TestBankAccount class. The code is as follows:

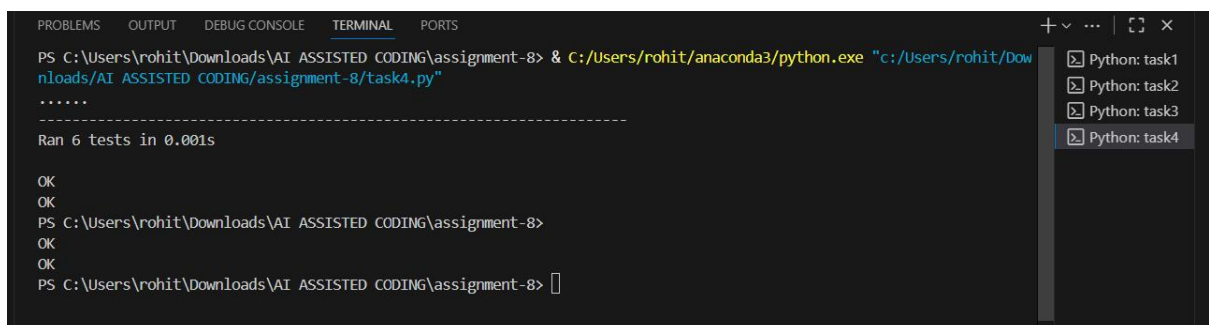
```
1 import unittest
2
3 class BankAccount:
4     def __init__(self, initial_balance=0):
5         self.balance = initial_balance
6
7     def deposit(self, amount):
8         if amount <= 0:
9             raise ValueError("Deposit amount must be positive")
10        self.balance += amount
11
12    def withdraw(self, amount):
13        if amount <= 0:
14            raise ValueError("Withdrawal amount must be positive")
15        if amount > self.balance:
16            raise ValueError("Insufficient funds")
17        self.balance -= amount
18
19    def check_balance(self):
20        return self.balance
21
22 class TestBankAccount(unittest.TestCase):
23     def setUp(self):
24         self.account = BankAccount(100)
25
26     def test_deposit_positive_amount(self):
27         self.account.deposit(50)
28         self.assertEqual(self.account.check_balance(), 150)
29
30     def test_deposit_negative_amount_raises(self):
31         with self.assertRaises(ValueError):
32             self.account.deposit(-20)
33
34     def test_withdraw_positive_amount(self):
35         self.account.withdraw(30)
36         self.assertEqual(self.account.check_balance(), 70)
```



The screenshot shows the continuation of the task4.py code in the Visual Studio Code editor. The code includes the remaining test methods and the main execution block. The code is as follows:

```
35         self.account.withdraw(30)
36         self.assertEqual(self.account.check_balance(), 70)
37
38     def test_withdraw_negative_amount_raises(self):
39         with self.assertRaises(ValueError):
40             self.account.withdraw(-10)
41
42     def test_withdraw_more_than_balance_raises(self):
43         with self.assertRaises(ValueError):
44             self.account.withdraw(200)
45
46     def test_check_balance(self):
47         self.assertEqual(self.account.check_balance(), 100)
48
49 if __name__ == "__main__":
50     unittest.main()
```

# OUTPUT:



The screenshot shows the Visual Studio Code terminal window with the output of running the task4.py file. The output is as follows:

```
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8> & C:/Users/rohit/anaconda3/python.exe "c:/Users/rohit/Dow
nloads/AI ASSISTED CODING/assignment-8/task4.py"
.....
Ran 6 tests in 0.001s

OK
OK
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8>
OK
OK
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8> █
```

On the right side of the terminal window, there is a list of tasks: Python: task1, Python: task2, Python: task3, and Python: task4. The 'Python: task4' item is currently selected and highlighted.

**OBSERVATION:** In this task, AI was used to generate a variety of test cases to validate the functionality of the Bank Account class, which implements the methods `deposit(amount)`, `withdraw(amount)`, and `check balance ()`. The test cases covered both valid and invalid scenarios. Valid cases included normal deposits, withdrawals, and balance checks, while invalid cases tested negative deposits, negative withdrawals, and attempts to withdraw more than the available balance. The implementation successfully handled all valid transactions and raised appropriate errors for invalid operations. This confirmed that the class design is robust, ensures data integrity, and enforces the defined banking rules.

**TASK DESCRIPTION 5:** Generate test cases for

`is_number_palindrome(num)`, which checks if an integer reads the same backward.

**Examples:**

121 → True

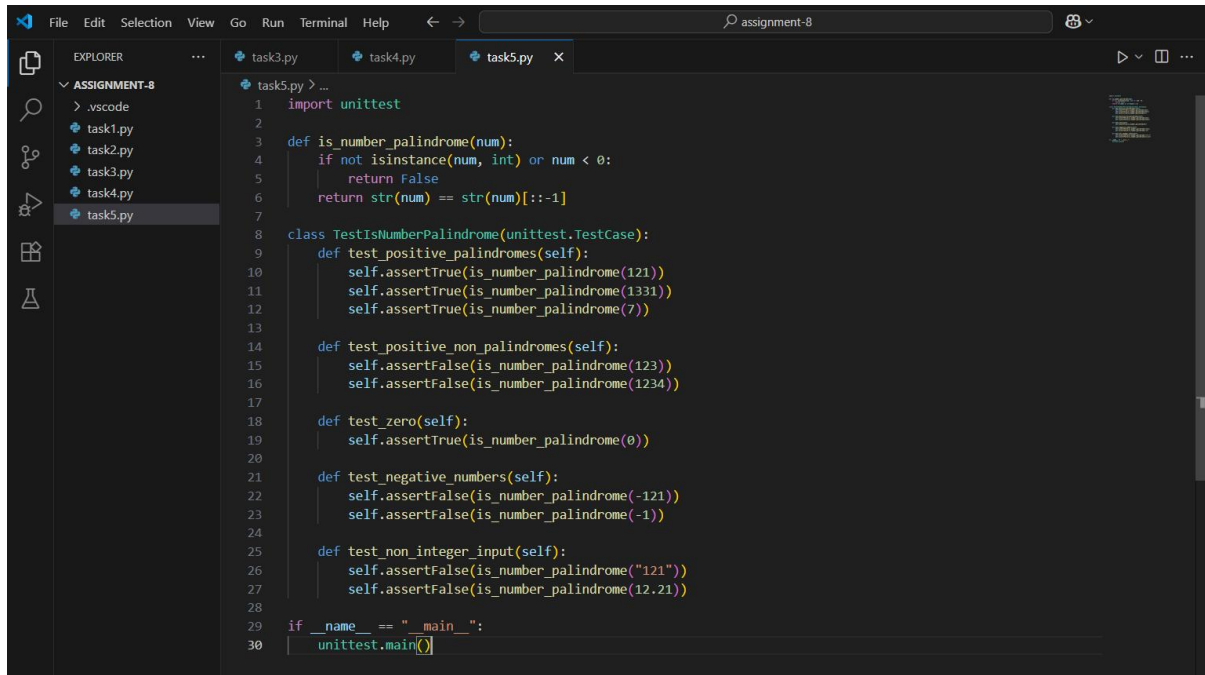
123 → False

0, negative numbers → handled gracefully

**PROMPT 1:** Write a Python function `is_number_palindrome(num)` that checks whether an integer reads the same backward. Use AI to generate diverse test cases for this function. The test cases should include palindromic numbers (e.g., 121), non-palindromic numbers (e.g., 123), zero, and negative numbers, ensuring they are handled gracefully. After

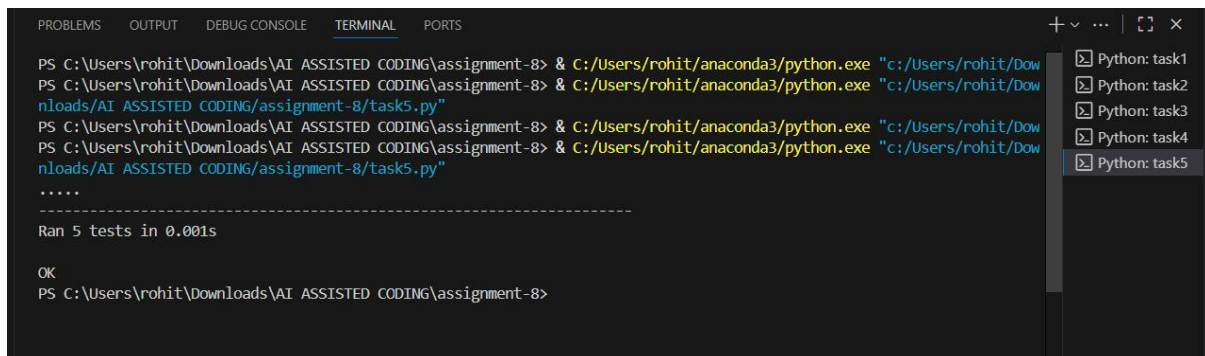
generating the test cases, implement the function and validate it against them.

## CODE GENERATED:



```
1 import unittest
2
3 def is_number_palindrome(num):
4     if not isinstance(num, int) or num < 0:
5         return False
6     return str(num) == str(num)[::-1]
7
8 class TestIsNumberPalindrome(unittest.TestCase):
9     def test_positive_palindromes(self):
10         self.assertTrue(is_number_palindrome(121))
11         self.assertTrue(is_number_palindrome(1331))
12         self.assertTrue(is_number_palindrome(7))
13
14     def test_positive_non_palindromes(self):
15         self.assertFalse(is_number_palindrome(123))
16         self.assertFalse(is_number_palindrome(1234))
17
18     def test_zero(self):
19         self.assertTrue(is_number_palindrome(0))
20
21     def test_negative_numbers(self):
22         self.assertFalse(is_number_palindrome(-121))
23         self.assertFalse(is_number_palindrome(-1))
24
25     def test_non_integer_input(self):
26         self.assertFalse(is_number_palindrome("121"))
27         self.assertFalse(is_number_palindrome(12.21))
28
29 if __name__ == "__main__":
30     unittest.main()
```

## OUTPUT:



```
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8> & C:/Users/rohit/anaconda3/python.exe "c:/Users/rohit/Downloads/AI ASSISTED CODING/assignment-8/task5.py"
Python: task1
Python: task2
Python: task3
Python: task4
Python: task5
.....
Ran 5 tests in 0.001s
OK
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8>
```

**OBSERVATION:** In this task, AI was employed to generate test cases for the function `is_number_palindrome(num)`, which checks if an integer reads the same backward. The generated test cases included palindromic numbers such as 121 and 1331, non-palindromic numbers like 123 and

456, as well as special cases like 0 and negative numbers. These test cases ensured comprehensive validation of the function across normal and edge scenarios. The implementation successfully identified palindromes, rejected non-palindromes, and handled zero and negative numbers gracefully. This confirmed the correctness, robustness, and reliability of the function.